Sobre la correcta manipulación de números aleatorios

Ángel García Baños*[1]
Fabio Guerrero**

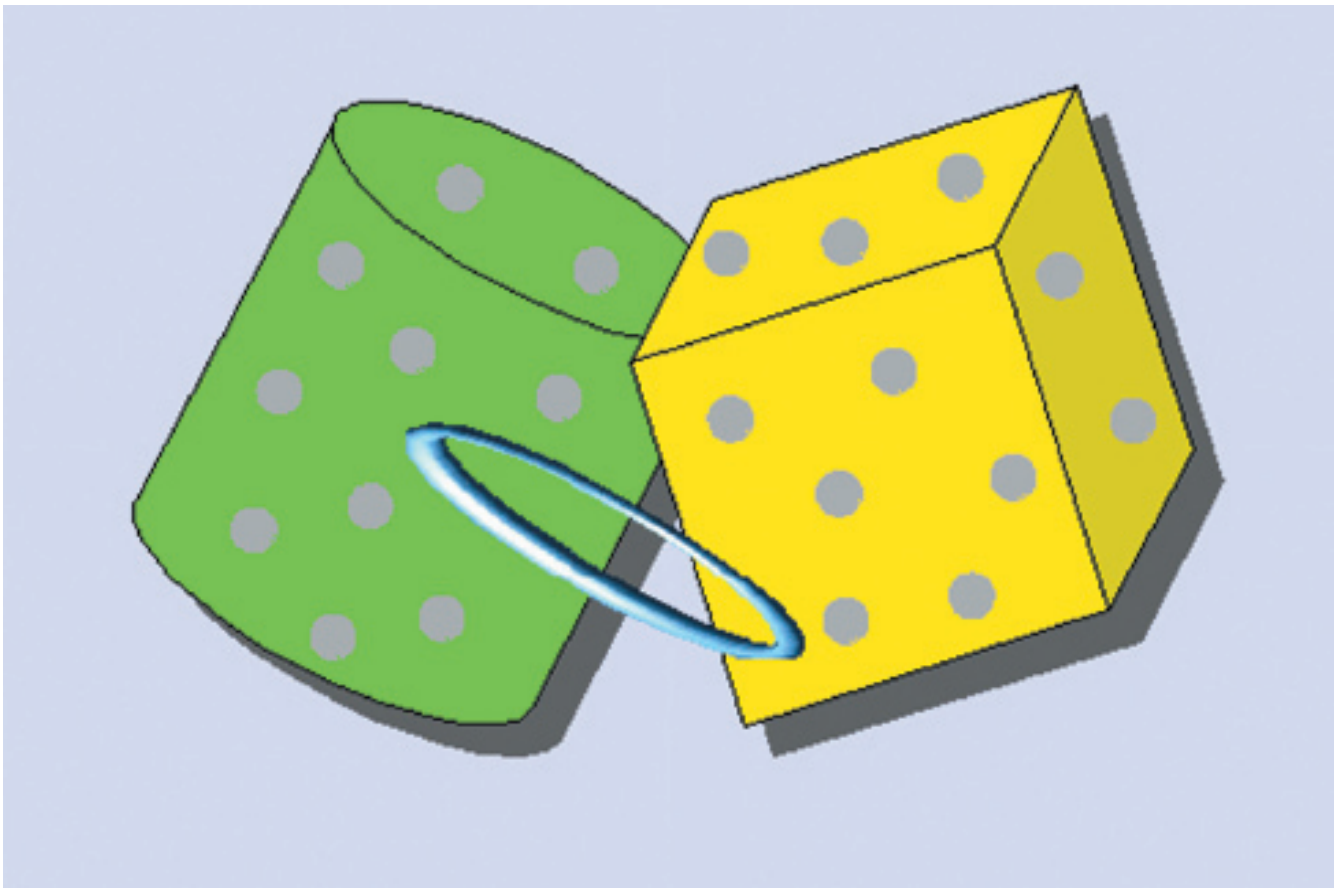


Imagen del autor

1 *Ph.D. Ingeniero de Telecomunicación por la Universidad Politécnica de Valencia y el de Ingeniero de Telecomunicación por la Universidad Politécnica de Madrid. Trabaja como profesor titular de la Escuela de Ingeniería de Sistemas y Computación de la Universidad del Valle. angarcia@pÉpÊp

** Ms.C. en Sistemas Electrónicos de Tiempo Real, Bradford University, United Kingdom, e Ingeniero en Electrónica y Telecomunicaciones de la Universidad del Cauca. Trabaja como profesor asistente de la Escuela de Ingeniería Eléctrica y Electrónica de la Universidad del Valle. Email: fguerrer@univalle.edu.co

Fecha de recepción:10/06/06 Fecha de aprobación:18/10/07

Resumen

Un generador de números aleatorios es algo precioso, sutil y delicado que conviene tratar con el máximo cuidado para no perder sus propiedades. En muchos casos se dispone de un generador de números seudoaleatorios con distribución de probabilidad uniforme y se desea adaptarlo para generar otro tipo de distribuciones. En este artículo se mostrarán los errores más habituales que se cometen al intentar hacerlo y también un método para evitarlos usando un par de ejemplos sencillos.

Palabras clave

Generación de números aleatorios, funciones de distribución de probabilidad, ruido.

Abstract

A random number generator is something precious, subtle, and delicate that it is worth to treat it carefully to avoid losing its properties. In many situations a pseudorandom number generator with uniform probability distribution is available and it is desired to adapt it for generating another kind of distributions. In this article the most common errors made trying to achieve this goal, as well as a method to avoid making these mistakes are presented using a pair of simple examples.

Key words

Random number generation, probability distribution functions, noise.

1. Introducción

Una secuencia aleatoria de números no se puede predecir. Su comportamiento se puede describir únicamente por sus propiedades estadísticas. Una secuencia seudoaleatoria posee las propiedades de una secuencia aleatoria durante un periodo de observación determinado. Sin embargo, no es completamente aleatoria, ya que a largo plazo se convierte en una señal determinística.

Cuando se está escribiendo un programa que necesite usar números generados al azar, en ANSI C y C++[1] existe la función:

```
int rand(void);
```

que genera números seudoaleatorios con distribución uniforme comprendidos entre 0 y RAND_MAX ambos incluidos. Funciones similares se puede encontrar en muchos otros entornos de programación. Usualmente RAND_MAX es el valor máximo positivo que se puede almacenar en un número entero (*int*). Las secuencias seudoaleatorias son deterministas pero difíciles de predecir si no se conoce la fórmula que las generó y el valor inicial de partida. El algoritmo es usualmente el método de congruencia lineal:

R[0] = semilla;

R[i] = (R[i−1] * B + 1) mod M; // Para i = 1,2,3... (Ec.1)

Para asegurarse de que la secuencia no se va a repetir, conviene cambiar el valor inicial cada vez que se vaya a usar en un programa. Esta inicialización se realiza con una "semilla" que sea distinta cada vez que se ejecute el programa. Para ello, existe también en ANSI C y C++ la función:

```
void srand(unsigned int seed);
```

Es común tomar como semilla el valor de la hora actual expresada en segundos, de la siguiente manera:

```
srand(time(0));
```

La sentencia anterior debe ejecutarse una única vez, al principio del programa. Un error muy común es inicializar varias veces, a lo largo del programa, el generador de números seudoaleatorios, creyendo que con ello va a conseguirse más impredecibilidad en la secuencia. En realidad se obtiene todo lo contrario, ya que la hora actual probablemente no cambie entre una llamada a time(0) y la siguiente, por lo que realmente lo que se está haciendo es volver al principio de la misma secuencia.

Estas secuencias son seudoaleatorias, ya que:

- se pueden predecir si se conoce la semilla.
- cuando la variable que mantiene el último valor generado se desborda la secuencia vuelve a repetirse . De todos modos, el periodo se puede hacer muy grande, eligiendo B y M adecuadamente en la (Ec.1).

Como se ve, es absolutamente imposible generar una secuencia estrictamente aleatoria por software. Ello solo puede lograrse empleando algún recurso de hardware. Por ejemplo, se puede digitalizar una fuente conocida de ruido como, verbigracia, el voltaje de codo inverso de un diodo zener. Pero no es habitual que un computador disponga de una tarjeta hardware específica generadora de ruido.

Sin embargo, hay formas más fáciles de obtener ruido en un computador estándar, tales como leer el ruido analógico muestreado en el micrófono de la tarjeta de sonido. Otra forma (empleada por el kernel de Linux) es usar un "recolector de entropía", que consiste en medir los intervalos entre eventos asíncronos, tales

como pulsaciones de teclas, movimientos del mouse, etc. Los datos aleatorios pueden leerse del dispositivo /dev/random o mejor de /dev/urandom. Desgraciadamente estas fuentes de ruido se demoran bastante en producir unos pocos bits realmente aleatorios. Por cuestiones de velocidad suele entonces recurrirse a una opción híbrida: obtener del hardware un número aleatorio, que se emplea como semilla para el generador de números seudoaleatorios por software.

2. Definición de aleatoriedad

Afirmar que una secuencia numérica dada es aleatoria, es absolutamente imposible, pues no hay una definición que capture la esencia de la aleatoriedad.[2] Sin embargo, existe una serie de propiedades que, de no cumplirse indican que la secuencia en cuestión no es aleatoria. Esas condiciones necesarias, pero no suficientes, de aleatoriedad de secuencias son:

1. Balance: Los símbolos presentes en la secuencia deben ser equiprobables. Para el caso binario, por la teoría de probablidades de grandes números, el número de ceros y unos en la secuencia debe diferir como máximo en 1. En términos de la teoría de la información, se puede considerar que la secuencia proviene de una fuente con máxima entropía.[3] Una prueba de verificación sencilla sería pasar la secuencia aleatoria por un compresor ideal y verificar que no existe ninguna ganancia por el efecto de la compresión.
2. Carrera: Una carrera se define como una secuencia de un mismo tipo de dígito. 1/2 de las carreras deben ser de longitud 1, 1/4 de longitud 2, 1/8 de longitud 3, etc. En general, la proporción de cualquier carrera de longitud n debe ser $1/2^n$.
3. Correlación: La autocorrelación de cualquier muestra debe ser cero para cualquier desplazamiento τ diferente de cero. Es decir, la secuencia no debe guardar el más mínimo parecido al compararse consigo misma, después de un desplazamiento de τ unidades ($\tau \neq 0$).

3. Algunas propiedades de los números aleatorios

Para entender los problemas que surgen con la incorrecta manipulación de números aleatorios, conviene reparar en las siguientes propiedades: [4]

1. Composición: Dos secuencias de números aleatorios con rangos $[0, 2^N-1]$ y $[0, 2^M-1]$ se pueden componer para formar una secuencia aleatoria más larga de rango $[0, 2^{N+M}-1]$ simplemente concatenándolas.
2. Extracción: Dada una secuencia de números aleatorios, si extraemos un bit individual, su secuencia también es aleatoria. En general, también lo es si se extrae cualquier grupo de bits.
3. La correlación de secuencias de bits extraídas de la manera indicada en la propiedad anterior es cero.
4. Decimación: si de una secuencia aleatoria de distribución uniforme se extrae un subconjunto en función de alguna propiedad ordinal (por ejemplo, los términos pares, etc) la secuencia resultante sigue siendo aleatoria y de distribución uniforme.
5. Selección: si una secuencia aleatoria de distribución uniforme se filtra aceptando o rechazando números en función de su magnitud, la nueva secuencia es también de distribución uniforme pero con un rango distinto (el rango seleccionado).

Un ejercicio acerca de la naturaleza de la aleatoriedad es el siguiente: ¿es la secuencia {0, 5, 4, 2, 9, 8, 6, 7, 3, 1} aleatoria? ¿cuál es su "ley de formación"?

En la Figura 1 se muestra la clase histograma (escrita en C++) y en la Figura 2 el programa principal para encontrar la función de distribución de un generador seudoaleatorio.

Figura 1. Clase histograma

```
class histograma
{
private:
int max_barras;
long valor_maximo;
long *barras;
public:
histograma(int m_b, long v_m);
~histograma() { delete [ ] barras;};
void add(long valor);
void resultados(ofstream &sal);
};

histograma::histograma(int m_b, long v_m)
{
 max_barras=m_b;
valor_maximo=v_m;
barras=new long[max_barras];
for(unsigned int i=0; i<max_barras; i++)
barras[i]=0;
}

void histograma::add(long valor)
{
if(valor<valor_maximo)
barras[MIN(max_barras−1,
(valor*max_barras)/valor_maximo)]++;
}

void histograma::resultados(ofstream &sal)
{
for(unsigned int i=0; i<max_barras; i++)
 {
sal << (long)(i*(valor_maximo/max_barras))<< "–"
<<(long)((i+1)*(valor_maximo/max_barras))
<< "\t" << barras[i] << endl;
 }
}
```

Figura 2. Programa principal

```
main()
{
histograma H(20, 20000);
for(long i=0; i<100000; i++)
// H.add(rand_exp(2000, 10000, 0.001));
H.add(rand_test(20000));

ofstream s;
s.open("resultados.txt");
H.resultados(s);
}
```

4. Construcción de generadores

4.1 Generador de rango más grande

Cuando se desea ampliar el rango de un generador de números aleatorios, un error muy común es multiplicar la salida por una constante:

$$X = K*rand();$$

El error está en que usando este método se producen "huecos" en la secuencia resultante. El número X tiene la predecible e indeseable propiedad de ser siempre múltiplo de K.

La mejor solución es emplear la propiedad 1:

$$X = K*rand() + rand\_K();$$

siendo rand_K() un generador de números con rango mas pequeño, [0, K).

4.2 Generador de rango más pequeño

Otro error muy común es emplear aritmética modular:

$$X = rand() \bmod M; \quad (Ec.2)$$

Este generador tiene problemas si M no es potencia de 2. En la Figura 3 se puede ver que la probabilidad de generar cada número no es uniforme.

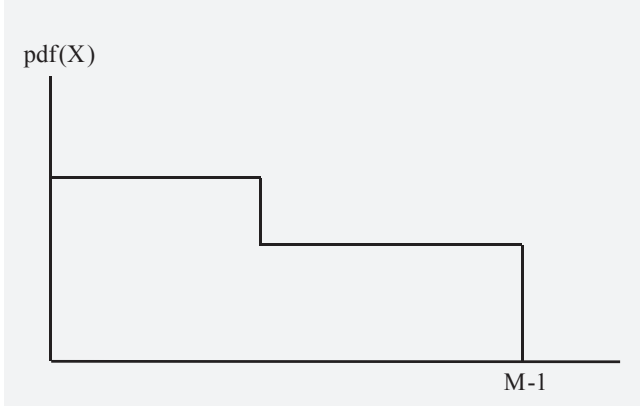


Figura 3. Distribución resultante usando aritmética modular

Una posible solución, que forma parte de la propuesta más general de este artículo es:

```
do {
    X=rand();
} while(x>=M);
```

En general, si se desea que el rango de salida sea $[N_{MIN}, N_{MAX}]$ el código sería:

```
do {
    x=N_MIN + rand();
} while(x>=N_MAX);
```

4.3 Generación de distribuciones arbitrarias continuas

Como se ha visto, un error muy común, (Ec. 2), es aplicar directamente la fórmula de distribución deseada sobre el generador de números aleatorios de distribución uniforme. Por ejemplo, para conseguir una distribución exponencial es bastante común (y equivocado) hacer:

```
X = min+1.0/exp(escala*(max–min)/
    (float)rand());
```

Ello conduce a sesgos en la calidad de la secuencia resultante (si hay una multiplicación por 2 faltarán los números impares, etc.). Si la fórmula es compleja, el sesgo puede ser difícil de detectar pero eso no quiere decir que no exista.

4.4 Ejemplo de un generador aleatorio con distribución exponencial

El problema se plantea así: dada una secuencia aleatoria de distribución uniforme S_U generar otra de distribución exponencial S_E. Se desea que S_E tanga un rango [min, max]. Se supone que el rango de S_U cubre por completo al de S_E. En caso contrario, siempre se puede aplicar la propiedad 1 para agrandar el rango.

La manera que se propone en este trabajo para evitar sesgos en la secuencia resultante S_E es la siguiente:

Se necesitan de partida dos secuencias aleatorias de distribución uniforme (S_A , S_B). Ello se consigue sin mas que decimar la secuencia de entrada S_U: los números de orden par por un lado y los de orden impar por otro:

$$S_A(k) = S_U(2*k)$$

$$S_B(k) = S_U(2*k+1)$$

Por la propiedad 4 ambas secuencias siguen siendo aleatorias y de distribución uniforme.

La primera secuencia, S_A, no se altera con ninguna operación matemática, sino que se filtra, eligiendo de

ella los números que caen dentro del rango deseado y descartando los demás. Por la propiedad 5 ello produce otra secuencia aleatoria de distribución uniforme en el nuevo rango.

La segunda secuencia, S_B, se compara con la función de probabilidad deseada (en este caso, exponencial) aplicada a la primera secuencia, S_A. Si está por encima, el correspondiente número de la secuencia S_A se acepta; y si no, se rechaza. Con ello se logra modular la probabilidad de aceptación, cambiando la distribución de uniforme a la función de probabilidad deseada.

A continuación se muestra una posible implementación en lenguaje C:

```
unsigned int rand_exp(unsigned int min, unsigned int
max, float escala)
{
 unsigned int n1;
float n2, limite;
do
 {
  do
   {
     n1=rand();
    } while(n1>(max−min));
  n2=rand()/(float)RAND_MAX;
  if(n1==0)
      limite=0.0;
   else
     limite=1.0/exp(escala*(max−min)/ (float)n1);
   } while(n2<limite);
return n1+min;
}
```

Obsérvese que aunque la función exponencial se aplica sobre la secuencia S_A, esto se hace solo con el fin de realizar una comparación posterior con S_B. La secuencia resultante S_E procede de decimar adecuadamente S_A, sin aplicar ningún tipo de operación matemática sobre ella, por lo que la secuencia de salida no tiene sesgos en su aleatoriedad.

4.5 Generación de distribuciones arbitrarias discretas

Una forma sencilla para generar por computador funciones discretas finitas con distribución arbitraria es crear un arreglo donde el índice está asociado a los símbolos s_i deseados Cada elemento del arreglo contiene la probabilidad acumulada asociada a su índice.

Se extrae un número al azar p de una fuente uniforme. El símbolo elegido es el símbolo que se produce al escoger el mínimo elemento tal que Arreglo[s_i] ≥ p.

Ejemplo. Se desea generar las letras A, B, C, D, E con probabilidades: P(A) = 0.498, P(B) = 0.327, P(C) = 0.125, P(D) = 0.030, y P(E) = 0.020

El arreglo contiene en este caso 5 elementos así:

A	B	C	D	E
498	825	950	980	1000

Se elige al azar un número p entre 1 y 1000 con probabilidad uniforme. Supongamos, por ejemplo, que p es igual a 923. El menor elemento del arreglo mayor que 923 es 950 y, por lo tanto, la fuente produciría la letra C. Este procedimiento se repite un número de veces suficientemente grande para caracterizar la fuente.

4.6 Generación de una distribución uniforme a partir de una distribución arbitraria

Von Neumann[5] planteó la idea de la simulación de una fuente uniforme a partir de una fuente sesgada (no equiprobable) con el siguiente ejemplo: lanzar una moneda cargada dos veces. Si sale cara-sello, generar un uno. Si sale sello-cara, generar un cero. En otro caso, empezar nuevamente.

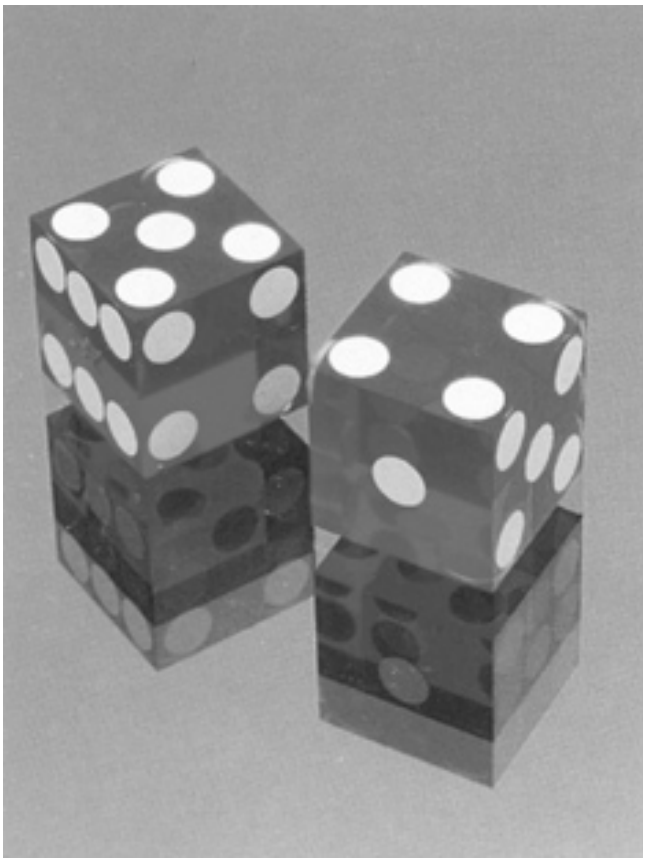
5. Conclusiones y observaciones

La idea básica sobre la manipulación correcta de números aleatorios radica en no intentar obtener la distribución deseada haciendo operaciones directamente sobre la fuente de aleatoriedad. En cambio se deben establecer criterios de descarte sobre la fuente aleatoria primaria dictados por la distribución aleatoria que se busca.

La secuencia propuesta como ejercicio es demasiado corta para chequear sobre ella las pruebas necesarias de aleatoriedad. Sin embargo, basta dedicarle unos minutos para comprobar que no existe una fórmula razonable que la pueda generar. La mayoría de la gente interpretaría esto como un fuerte indicio de su aleatoriedad.

Sin embargo, nada más lejos de la realidad: la secuencia está en orden alfabético: cero, cinco, cuatro, dos, nueve, ocho, seis, siete, tres, uno. Es decir, la secuencia no es en absoluto aleatoria.

Resulta bastante sorprendente comprobar que en idioma inglés (8, 5, 4, 9, 1, 7, 6, 3, 2, 0) también parece



razonablemente aleatoria, para quién no haya descubierto la "fórmula".

Este sencillo ejemplo nos ilustra cómo el concepto de aleatoriedad depende fuertemente de la inteligencia y del tipo de inteligencia del humano (o ente) que investiga la secuencia. Y, aunque no sea obvio, también depende de la persona o ente que generó la secuencia.

Bibliografía

1. Herbert Schildt, "C++. Manual de Referencia", Osborne McGraw–Hill, 1995.
2. Papoulis, A. *Probability, Random Variables, and Stochastic Processes*. New York: McGraw-Hill, 1991, pp. 221-237.
3. Wells R. B., Applied Coding and Information Theory for Engineers. Upper Saddle River, New Jersey: Prentice Hall, 2002, ch. 1.
4. J. M. Noras, Comunicación Personal, "Pseudo–random Sequence Generators: Linear Feedback Shift Registers, Cellular Automata and Carry Feedback Shift Registers", Bradford BD7 1DP, Department of Electronic and Electrical Engineering, UK.
5. J. Von Neumann J., "Various techniques used in connection with random digits", Nat. Bur. Stand., Appl. Math Ser., vol. 12, pp. 36 38, 1951.